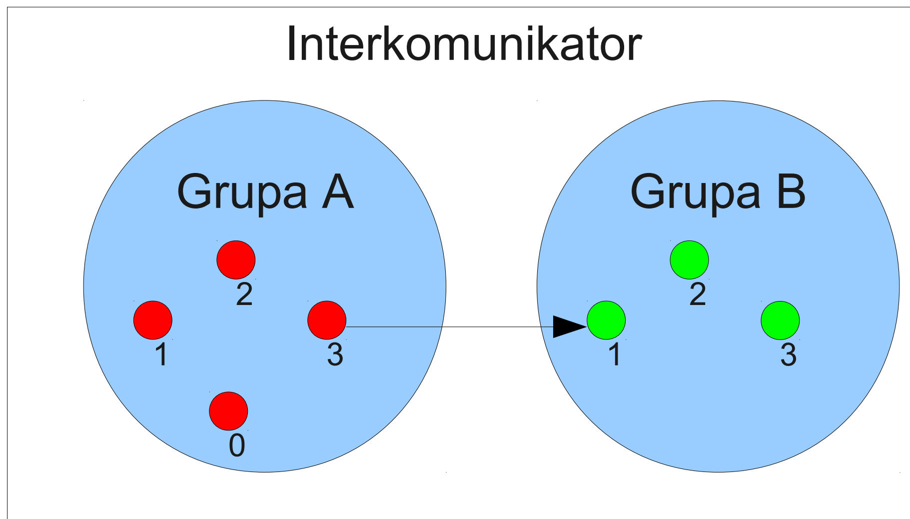
Rozszerzenia MPI-2

Dynamiczne tworzenie procesów

- Aplikacja MPI – 1 jest statyczna z natury
  - Liczba procesów określana jest przy starcie aplikacji i się nie zmienia.
  - Gwarantuje to szybką komunikację procesów.
  - Ładnie współpracuję z menadżerami zasobów (np. PBS/TORQUE).
- Kiedy `MPI_Init` powraca, wszystkie procesy wystartowały i są elementami komunikatora `MPI_COMM_WORLD`.
- Każdy proces może się komunikować z każdym innym procesem aplikacji używając MPI_COMM_WORLD.
- MPI-2 dostarcza mechanizmy do
  - ***Kolektywnego*** tworzenia nowych procesów.
  - Połączenia dwóch aplikacji stworzonych niezależnie
- Dynamicznego tworzenie i łączenie procesów wykorzystuje intensywnie pojęcie interkomunikatora – zdefiniowane już w MPI-1, ale rzadko wykorzystywane.

Interkomunikator w MPI



- Z punktu widzenie procesu wysyłającego grupa lokalna to A, zdalna B, a procesu odbierającego grupa lokalna to B, zdalna A

- Interkomunikator składa się z dwóch grup procesów.
- Grupa której proces jest członkiem, to ***grupa lokalna*** (dla tego procesu) a której proces nie jest członkiem to ***grupa zdalna***.
- Wszystkie operacje Send/Recv odbywają się pomiędzy procesami różnych grup. Numer procesu nadawcy (operacja Send) oraz odbiorcy (operacja Recv) jest numerem w grupie zdalnej.
- `MPI_Comm_size` na interkomunikatorze zwraca rozmiar grupy lokalnej, `MPI_Comm_remote_size` – grupy zdalnej.
- Operacje grupowe w MPI-1 niezdefiniowane, w MPI-2 dostępne (np. MPI_Bcast jeden proces z grupy lokalnej wysyła informacje wszystkim procesom grupy zdalnej)

Operacja MPI_Comm_spawn

- `MPI_Comm_spawn(char *command,char *argv[], int maxprocs,MPI_Info info,int root, MPI_Comm comm, MPI_Comm *intercomm, int errorcodes[])`
- Tworzy `maxprocs` ***potomnych*** procesów MPI, przy czym każdy z nich powstaje w wyniku uruchomienie programu `command`. `argv` zawiera argumenty programu (wiersz polecenia), przy czym w odróżnieniu od konwencji języka C, `argv[0]` nie jest nazwą programu
- Jest to operacja kolektywna, która jest wykonywana przez wszystkie procesy ***rodzicielskie*** w komunikatorze `comm` (np. `MPI_COMM_WORLD`).
- Tylko proces o numerze `root` musi dostarczyć argumenty na lewo od `root`.
- Wynikiem jest interkomunikator z punktu widzenia procesów rodzicielskich grupa lokalna składa się z procesów rodzicielskich a zdalna z potomnych.
- Interkomunikator ten może być pobrany przez procesy potomne funkcją `int MPI_Comm_get_parent(MPI_Comm *parent)`. Oczywiście z punktu widzenia procesów potomnych grupa zdalna składa się z procesów rodzicielskich a lokalna z potomnych.
- Procesy potomne mają swój własny komunikator MPI_COMM_WORLD.

Obiekty typu MPI_Info

- Służą do przekazywania funkcjom MPI-2 dodatkowych, niestandardowych informacji, wpływających na działanie funkcji i przekazujących informacje zależne od implementacji MPI oraz środowiska (menadżer zasobów, system operacyjny).
- Obiekt jest tworzony funkcją int `MPI_Info_create(MPI_Info *info)` a zwalniany funkcją int `MPI_Info_free(MPI_Info *info)`.
- Obiekt jest kolekcją par (klucz,wartość)
- Funkcja `MPI_Info_set(MPI_Info info, char *key,char *value)` dodaje do obiektu info parę `(key,value)`.
- Wartość domyślna `MPI_INFO_NULL`.
- Klucze w implementacji OpenMPI
  - host - wartość jest nazwą maszyny, na której uruchomić procesy.
  - hosts - nazwa pliku z nazwami maszyn.
  - path - ścieżka do pliku wykonywalnego.
  - wdir - katalog roboczy do pliku wykonywalnego.

Łączenie niezależnie uruchomionych aplikacji MPI - strona serwera

- Mechanizm w MPI oparty jest na modelu klient-serwer i przypomina interfejs gniazd protokołu TCP.
- Funkcja `int MPI_Open_port(MPI_Info info, char *port_name)` tworzy nazwę portu na którym serwer będzie odbierał połączenia. Funkcja `int MPI_Close_port(char *port_name)` zamyka otwarty port.
- Nazwę portu należy „ręcznie” przekazać klientom.
- Funkcja `int MPI_Comm_accept(char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm)` tworzy połączenie aplikacji serwera z aplikacją klienta.
  - Jest wywoływana kolektywnie przez wszystkie procesy w komunikatorze `comm` (np. `MPI_COMM_WORLD`).
  - Tworzy interkomunikator do komunikacji z klientem i zapisuje go pod adresem `newcomm`
  - Argumenty `info` i `port_name` są dostarczane tylko przez proces o numerze `root`.
- Rozłączenie serwera i klienta przy pomocy funkcji `int MPI_Comm_disconnect (MPI_Comm *comm)`. wywołanej na interkomunikatorze otrzymanym w wyniku `MPI_Comm_accept`.

Łączenie niezależnie uruchomionych aplikacji MPI - strona klienta

- Funkcja `int MPI_Comm_connect(char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm)` jest wykonywana kolektywnie przez procesy klienta na komunikatorze `comm`.
- Wykonuje połączenie z serwerem nasłuchującym na porcie `port_name`, tworzy interkomunikator do komunikacji z serwerem i zapisuje go pod adresem `newcomm`.
- Nazwę portu należy jakoś przekazać klientowi od serwera.
- Klient rozłącza się również przy pomocy funkcji .
- W następującym przykładzie
  - Serwer składa się z jednego procesu. Przetwarza dane przesłane przez klienta do momentu gdy odbierze komunikat o etykiecie 1. Komunikat o etykiecie zero kończy pracę serwera.
  - Klient pobiera nazwę portu z wiersza poleceń.

Przykład: serwer (1)

```
#include "mpi.h"
int main( int argc, char **argv )
{
    MPI_Comm client;
    MPI_Status status;
    // MAX_PORT_NAME to maksymalna długość nazwy portu
    char pname[MPI_MAX_PORT_NAME];
    double buf[MAX_DATA];
    int size, again;

    MPI_Init( &argc, &argv );
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (size != 1) MPI_Abort(MPI_COMM_WORLD,1);
    MPI_Open_port(MPI_INFO_NULL, pname);
    printf("server available at %s\n",pname);
    while (1) {
        // Nawiąż połączenie z klientem
        MPI_Comm_accept(pname,MPI_INFO_NULL,0,MPI_COMM_WORLD,
                                 &client );
        again = 1;
```

Przykład: serwer (2)

```
        while (again) {
                // Odbierz komunikat od dowolnego procesu klienta
                 MPI_Recv( buf, MAX_DATA, MPI_DOUBLE,
                  MPI_ANY_SOURCE, MPI_ANY_TAG, client,&status );
              switch (status.MPI_TAG) {
                   // Zakończ pracę serwera
                   case 0: MPI_Comm_free( &client );
                           MPI_Close_port(port_name);
                           MPI_Finalize();
                           return 0;
                   // Rozłącz się z klientem
                   // i akceptuj połączenia na nowo
                   case 1: MPI_Comm_disconnect( &client );
                           again = 0;
                           break;
                   case 2: // zrób coś z komunikatem
                   }
              }
        }
}
```

Przykład: klient

```
#include "mpi.h"
int main( int argc, char **argv )
{
    MPI_Comm server;
    double buf[MAX_DATA];
    char port_name[MPI_MAX_PORT_NAME];
    MPI_Init( &argc, &argv );
    // Nazwa portu przekazana przez wiersz polecenia
    strcpy(port_name, argv[1] );
    // Połącz się z serwerm tworząc interkomunikator
    MPI_Comm_connect( port_name, MPI_INFO_NULL, 0,
         MPI_COMM_WORLD,&server );

    while (!done) {
        tag = 2;
        MPI_Send( buf, n, MPI_DOUBLE, 0, tag, server );
        done = .......
        }
    MPI_Send( buf, 0, MPI_DOUBLE, 0, 1, server );
    MPI_Comm_disconnect( &server );
    MPI_Finalize();
}
```

Publikowanie nazw

- Poprzedni przykład był dość niewygodny, ponieważ nazwę portu wydrukowaną przez serwer trzeba było ręcznie wpisać do wiersza poleceń, aby przekazać ją klientowi.
- Interfejs publikowania nazw pozwala na publikację nazwy portu przez serwer pod nazwą usługi przy pomocy funkcji `int MPI_Publish_name(char *service_name, MPI_Info info, char *port_name)`. Publikacja obowiązuje aż do wywołania `int MPI_Unpublish_name(char *service_name, MPI_Info info, char *port_name)`.
- Nazwa portu może być uzyskana przez klienta przy pomocy funkcji: `int MPI_Lookup_name(char *service_name, MPI_Info info, char *port_name)`.
- Problem polega na zapewnieniu widoczności publikowanych nazw poza obręb aplikacji serwera.

Ublikowanie nazw - przykład

- Strona serwera

```
MPI_Open_port(MPI_INFO_NULL, port_name);
MPI_Publish_name("ocean", MPI_INFO_NULL, port_name);
MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF,
&intercomm);

/* Zrób coś z interkomunikatorem intercomm */

MPI_Unpublish_name("ocean", MPI_INFO_NULL, port_name);
```

- Strona klienta

```
MPI_Lookup_name("ocean", MPI_INFO_NULL, port_name);
MPI_Comm_connect( port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF,
                      &intercomm);
```

Plikowe wejście/wyjście

- Stanowiło problem w Państwa projekcie. (w programie Mandelbrot).
- Rozwiązanie w postaci jednego centralnego procesu odbierającego dane od pozostałych i zapisującego je do pliku bardzo źle się skaluje.
- Inne rozwiązania np. każdy proces zapisuje dane do swojego pliku też nie wyglądają zachęcająco.
- Potrzebne jest rozwiązanie pozwalające na równległy zapis przez wiele procesów do wspólnego pliku.
- Rozwiązanie powinno wspierać istniejące równoległe systemy plików (n.p. PIOFS oraz GPFS IBMa, GFS firmy RedHat, ...)
- Uwaga: Funkcje wejścia wyjścia w MPI są dla nieformatowanych plików binarnych.

Otwarcie pliku

- `int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh)`
- Otwarcie to operacja kolektywna dla wszystkich procesów w komunikatorze `comm`.
- `filename` to ścieżka do pliku, a `info` to wskazówki dla implementacji.
- `amode` to tryb otwarcia; jedna ze stałych lub suma bitowa:
  - MPI_MODE_RDONLY tylko do odczytu
  - MPI_MODE_RDWR odczyt i zapis
  - MPI_MODE_WRONLY tylko zapis
  - MPI_MODE_CREATE stwórz plik jeśli nie istnieje
  - MPI_MODE_EXCL błąd jeżeli próbujesz stworzyć plik który istnieje
  - MPI_MODE_DELETE_ON_CLOSE usuwa plik po zamknięciu
  - MPI_MODE_UNIQUE_OPEN nikt inny nie otworzy pliku,
  - MPI_MODE_SEQUENTIAL dostęp wyłącznie sekwencyjny
  - MPI_MODE_APPEND wszystkie wskaźniki do pliku ustaw na koniec.
- Pod adresem `fh` zostanie zapisany uchwyt do pliku, który jest podawany we wszystkich operacjach na nim.

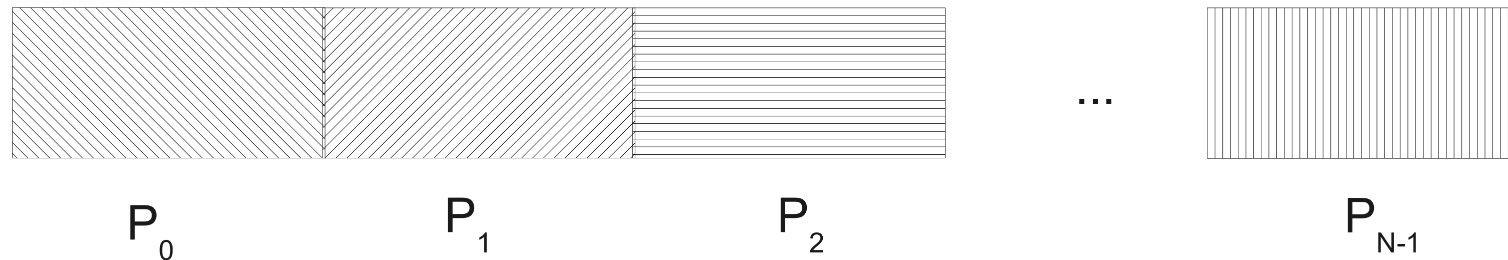
Zamknięcie pliku i kilka innych operacji.

- Funkcja `int MPI_File_close(MPI_File *fh)` zamyka plik.
- Zamknięcie jest również operacją kolektywną. (muszą wywołać je wszystkie procesy w komunikatorze)
- Nie podajemy komunikatora, bo komunikator jest przechowywany w obiekcie MPI_File.
- Wszystkie pliki muszą być zamknięte przed wykonaniem.
- Funkcja int `MPI_File_delete(char *filename, MPI_Info info)`, będąca operacją kolektywną usuwa plik o nazwie `filename`.
- Funkcja int `MPI_File_get_size(MPI_File fh, MPI_Offset *size)` zapisuje pod adresem `size` rozmiar pliku.
- Funkcja int `MPI_File_preallocate(MPI_File fh, MPI_Offset size)` będąca operacją kolektywną zwiększa rozmiar pliku do `size` jednocześnie alokując przestrzeń w pamięci masowej dla zwiększonego pliku.

Odczyt/zapis indywidualny

- `int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)` wykonuje odczyt `count` elementów typu `datatype` do bufora o adresie `buf` zapisując status tej operacji pod adresem `status`.
- `int MPI_File_write(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)` wykonuje zapis do pliku.
- Funkcje te ***nie są operacjami kolektywnymi !!!*** Odczyt/zapis jest wykonany przez każdy proces indywidualnie.
- Semantyka podobna jak w przypadku wywołań systemowych Linuksa.
- W MPI każdy proces ma własny wskaźnik bieżącej pozycji w pliku. Może on być zmieniany na wartość `offset` przy pomocy funkcji `int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)`, gdzie parametr `whence` może przyjąć jedną z trzech wartości:
  - MPI_SEEK_SET ustaw wskaźnik od początku pliku
  - MPI_SEEK_CUR ustaw wskaźnik od bieżącej pozycji wskaźnika
  - MPI_SEEK_END ustaw wskaźnik od końca pliku

Przykład



- Mamy N procesów, każdy chce odczytać blok bajtów (o identycznej długości).
- Bloki bajtów poszczególnych procesów są umieszczone jeden po drugim.
- Jeżeli długość bloku jest równa M, to proces o numerze i odczytuje bajty od numeru i*M do i*M+M-1.
- Czy ten przykład coś Państwu przypomina ???.

Kod programu

```
#include "mpi.h"
#define FILESIZE (1024 * 1024)

int main(int argc, char **argv)
{
    int *buf, rank, nprocs, nints, bufsize;
    MPI_File fh; MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    bufsize = FILESIZE/nprocs; //rozmiar bloku pliku
    buf = (int *) malloc(bufsize);
    nints = bufsize/sizeof(int); //liczba elementów typu MPI_INT
    MPI_File_open(MPI_COMM_WORLD, "datafile",MPI_MODE_RDONLY,
                  MPI_INFO_NULL, &fh);
    MPI_File_seek(fh, rank*bufsize, MPI_SEEK_SET);
    MPI_File_read(fh, buf, nints, MPI_INT, &status);
    MPI_File_close(&fh);
    free(buf);
    MPI_Finalize();
    return 0;
}
```

Co robić w sytuacji gdy każdy proces chce otworzyć inny plik ?

- Otwieranie kolektywne n plików przez n procesów na komunikatorze `MPI_COMM_WORLD` nie ma sensu
- Użyć ulubionego interfejsu we/wy (`fopen`, etc.)
- Użyć predefiniowanego komunikatora `MPI_COMM_SELF` – jest to komunikator którego członkiem jest jedynie aktualny proces.

Nieblokujące operacje plikowego we/wy

- Funkcje `int MPI_File_iread(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)` oraz `int MPI_File_iwrite(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)` wykonują nieblokujący odczyt/zapis do pliku
- Funkcje inicjalizują odczyt/zapis, po czym pod adresem `request` tworzą obiekt żądania identyfikujący trwającą operację odczytu/zapisu.
- Na obiekcie żądania może być wykonana funkcja `MPI_Wait` wstrzymująca wołający proces do momentu zakończenia operacji (Uwaga: ***zakończenie operacji nie oznacza że dane są na dysku !*** - służy do tego funkcja `MPI_File_sync` lub `MPI_Test` sprawdzająca czy operacja się zakończyła bez blokowania procesu.
- Pozwalają na nakładanie się obliczeń i operacji plikowego we/wy:

```
MPI_Request request;
MPI_Status &status;
MPI_File_iwrite(fh,buf,count,MPI_DOUBLE,&request)

/* Teraz wykonaj obliczenia !!! */

MPI_Wait(&request,&status);
```

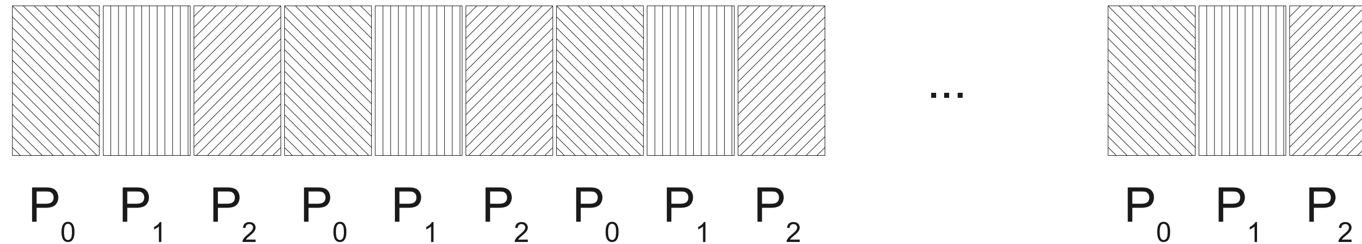
Odczyt i zapis od określonej pozycji z pliku

- `int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status).`
- `int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status).`
- Dodatkowy parametr `offset` specyfikuje pozycje w pliku, od której ma nastąpić odczyt.
- Należy stosować te funkcje zamiast pary `MPI_File_seek`+funkcja odczytująca/zapisująca w aplikacjach wielowątkowych (kto wie dlaczego ???).
- Istnieją również wersje nieblokujące: `MPI_File_iwrite_at` oraz `MPI_File_iread_at`.

Operacje kolektywne

- Funkcje odczytu/zapisu omówione dotychczas ***nie są operacjami kolektywnymi***.
- A zatem możliwa jest sytuacja w której plik otwierany jest kolektywnie przez wszystkie procesy w komunikatorze MPI_COMM_WORLD a następnie zapis/odczyt jest wykonywany przez jeden proces.
- Funkcje `int MPI_File_read_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)` oraz `int MPI_File_write_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)` służą do przeprowadzania kolektywnej operacji wejścia wyjście przez wszystkie procesy w komunikatorze.
- W przypadku operacji kolektywnej implementacja wie, że inne procesy również wykonują operację, co pozwala na przeprowadzenie szeregu optymalizacji.
- W naszym przykładzie zastosowanie operacji kolektywnych może prowadzić do wzrostu wydajności.
- Funkcje `MPI_File_read_at` oraz `MPI_File_write_at` mają również swoje kolektywne wersje.

Co robić w sytuacji następującego dostępu



- Każdy proces ma do zapisania n bloków do pliku (Ciekawe, czy to coś Państwu przypomina ?)
- Wyjście najprostsze n operacji dyskowych przez każdy proces.
- Wyjście skomplikowane: Każdy proces definiuje nieciągły ***widok*** pliku i zapis jedną operacją kolektywną – może prowadzić do znacznie większej wydajności na odpowiednim sprzęcie i implementacji MPI.

To wszystko na dzisiaj

- Pominęliśmy widoki plików

- Oraz komunikację jednostronną (one-sided communication).

- Życzę powodzenia na zaliczeniu.